Unreliable Heterogeneous Workers in a Pool-based Evolutionary Algorithm

TRACK:

ABSTRACT

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program synthesis*

General Terms

Performance,Theory,Experimentation

Keywords

1. INTRODUCTION

Thanks in part to evolutionary computation (EC) research, scientists and engineers from many fields now understand the remarkable power of the natural search process described by the biological theory of Neo-Darwinian evolution []. Inspired in biological evolution, EC researchers have developed a variety of search and optimization algorithms [].

While EAs are inspired by evolution, they mostly follow an abstract model of the natural process. For instance, one aspect that is omitted from most EAs is an open-ended search process, in practice EAs are used to solve problems with well defined objectives, while natural evolution is an adaptive process without an a priori goal or purpose. Such open-ended EAs have been developed, mostly for artistic design [], interactive evolution [] and artificial life systems []. Other interesting features of biological evolution, is that it is an intrinsically parallel, distributed and asynchronous process, undoubtedly important features that have allowed evolution to produce impressive results throughout nature. However, some of these features are not trivially included into standard EAs, which are mostly coded as sequential and synchronous algorithms [].

For instance, a large body of work exists in EA parallelization, using multiple CPU cores [], PCs [] and GPUs []. However, distributed EAs have started to become common only recently []. In particular, recent trends in information technology have opened new lines of future development for EC research. Today, computing resources computing power range from personal computers and smart-devices to massive data centers. These resources are easily accessible through popular internet technologies, such as cloud computing, peer-to-peer (P2P), and web environments.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GECCO'14, July 12-16, 2014, Vancouver, BC, Canada.
Copyright 2014 ACM TBA ...$15.00.

Moreover, these technologies are intended for the development of parallel, distributed and asynchronous systems, such that an EA developed on top of them could easily reap the benefits of these features. As stated before, several EAs have been proposed that distribute the evolutionary process among heterogeneous devices, not only among controlled nodes within an in-house cluster or grid, but also to others outside the data center, in users' web browsers, smart phones or external cloud based virtual machines. This reach out approach allows researchers to use low cost computational power that would not be available otherwise, but on the other hand, have the challenge to manage heterogeneous unreliable computing resources. In particular, we are interested in systems that follow a pool-based approach, where the search process is conducted by a collection, of possibly heterogeneous, collaborating processes using a shared repository or population pool. We will refer to succh algorithms as Pool-based EAs or PEAs, and highlight the fact that such systems are intrinsically parallel, distributed and asynchronous.

Despite promising results, PEAs present several notable challenges. From a technological perspective, for example, lost connections, low bandwidth, abandoned work, security and privacy are all important issues that must be addressed. However, here we focus on a common issue with most EAs, that is only amplified in a PEA, a problem that can be referred to as parametrization. In general, among machine learning methodologies, EAs are highly criticized by the large number of parameters they posses, that for real world problems need to be tuned empirically or require additional heuristic processes to be included into the search to adjust the parameters automatically []. In the case of a PEAs, this issue is magnified since the underlying system architecture adds several degrees of freedom to the search process.

This work studies a recently proposed pool-based system called EvoSpace, a framework to develop PEAs using a heterogeneous collection of possibly unreliable resources. EvoSpace is based on Linda's tuple space [**?**] coordination model, where each node asynchronously pulls its work from a central shared memory. The core elements of EvoSpace are a central repository for the evolving population and remote clients here called EvoWorkers which pull random samples of the population to perform on them the basic evolutionary processes (selection, variation and survival), once the work is done, the modified sample is pushed back to the central population. Despite some promising initial results [], research devoted to EvoSpace has not addressed the parametrization issue mentioned above. In the study presented here, a recent approach called Randomized Parameter Setting Strategy (RPSS) [] is applied to EvoSpace and tested on several benchmark problems. The idea behind RPSS is that in a distributed EA, algorithm parametrization may be completely skipped for a successful search, with research showing that when the number of distributed process is large enough, algorithm parameters can be set randomly and still achieve good overall results. However, work on RPSS has only focused on the well-known Island Model for EAs, a distributed but synchronous system. On the other hand, the goal of this work is to evaluate RPSS on a complete PEA implemented through EvoSpace.

The remainder of the paper proceeds as follows. Section 2

2. RELATED WORK

Given the computational cost of many EAs, researchers have turned towards internet and cloud based software systems. For instance, Fernández et al. []use the well-known Berkeley Open Infrastructure for Network Computing (BOINC) to distribute EA runs across a heterogeneous network of volunteer computers using virtual machines.

However, such a system, as well as others distributed or parallel systems, does not follow the PEA approach studied in the current paper. In general, a pool-based system employs a central repository (real or virtual) where the evolving population is stored. Distributed clients interact with the pool, performing some or all of the basic EA processes (selection, genetic operators, survival). For example, Smaoui et al. [**?**] uses BOINC redundancy to deal with the volatility of nodes and unreliability of their results. In this work each BOINC work unit consisted of a fitness evaluation task and multiple replicas were produced and sent to different clients, later when outputs were received a validation step ensured all outputs match.

Merelo et al. [**?**] developed a Javascript PEA that distributes the evolutionary process over the web, this provides the added advantage of not requiring the installation of additional software in each computing node. In this work the server receives an Ajax request with the best individual obtained from the local evolution in clients, and then responds with additional parameters and the best individual in the population so far. Other similar cloud-based solutions are based on a global queue of tasks and a Map-Reduce implementation which normally handles failures by the re-execution of tasks [**?**, **?**, **?**].

In general, this pool-based approach can be traced back to the A-Teams system [**?**], which is not restricted to evolutionary algorithms. Another proposal is made by G. Roy et al. [**?**], who developed a multi-threaded system with a shared memory architecture that is executed within a distributed environment and achieves substantial performance gains compared to standard approaches. On the other hand, Bollin and Piastra [**?**] emphasize persistence over performance, proposing a system that decouples population storage from the basic evolutionary operations. A similar decoupled model is proposed by Merelo et al. [**?**], who used a database to store the population that is accessed through a web-server. Finally, a recent PEA is SofEA proposed by Merelo et al. [**?**, **?**], an EA that is mapped to a central CouchDB object store. It provides an asynchronous and distributed search process, where the four main evolutionary operators are decoupled from the evolving population.

This work, however, focuses on EvoSpace, a framework to develop PEAs using heterogeneous and unreliable resources [], described in greater detail in the following section.

3. EVOSPACE

EvoSpace is based on Linda's tuple space [**?**] coordination model, where each node asynchronously pulls its work from a central shared memory. The core elements of EvoSpace are a central repository for the evolving population and remote clients called EvoWorkers, which pull random samples of the population to perform on them the basic evolutionary processes (selection, variation and survival), once the work is done, the modified sample is pushed back to the central population.

This model contrasts with the use of a global queue of tasks and implementations of map-reduce algorithms, favored in other proposals PEAs [**?**, **?**, **?**]. Following the tuple space model, when individuals are pulled from the EvoSpace container these are removed from it, so that no other EvoWorker can use them. This design decision has several known benefits relevant to concurrency control in distributed systems, and also is an effective way of distributing the workload. Leaving a copy of the individual in the population server free to be pulled by other EvoWorkers will result in redundant work and this could be costly if the task at hand is time consuming. EvoWorkers are expected to be unreliable, as they can loose a connection or are simply shut down or removed from the client. When an EvoWorker is lost, so are the individuals pulled from the repository. Depending on the type of algorithm been executed, the lost of these samples could have a high cost. To address the problem of unreliable EvoWorkers, EvoSpace uses a simple re-insertion algorithm that also prevents the starvation of the population pool. Other pool based algorithms normally use a random insertion technique, but we argue this could negatively impact the outcome of the algorithm in some cases.

EvoSpace [*references hidden for blind review*] consists of two main components (see figure 1): (i) the EvoSpace container that stores the evolving population and (ii) EvoWorkers, which execute the actual evolutionary process, while EvoSpace acts only as a population repository. In a basic configuration, EvoWorkers pull a small random subset of the population, and use it as the initial population for a local EA executed on the client machine. Afterwards, the evolved population from each EvoWorker is returned to the EvoSpace container. When individuals are pulled from the container they remain in a phantom state, they cannot be pulled again but they are not deleted. Only if and when the EvoWorker returns the replacement sample phantoms are truly deleted. If the EvoSpace container is at risk of starvation or optionally when a time-out occurs new phantom individuals are re-inserted to the population and available again. This can be done because a copy of each sample is stored in a priority queue used by EvoSpace to re-insert the sample to the central population; similar to games where characters are Respawned after a certain time. In the experiments conducted in this work re-insertion occurs when

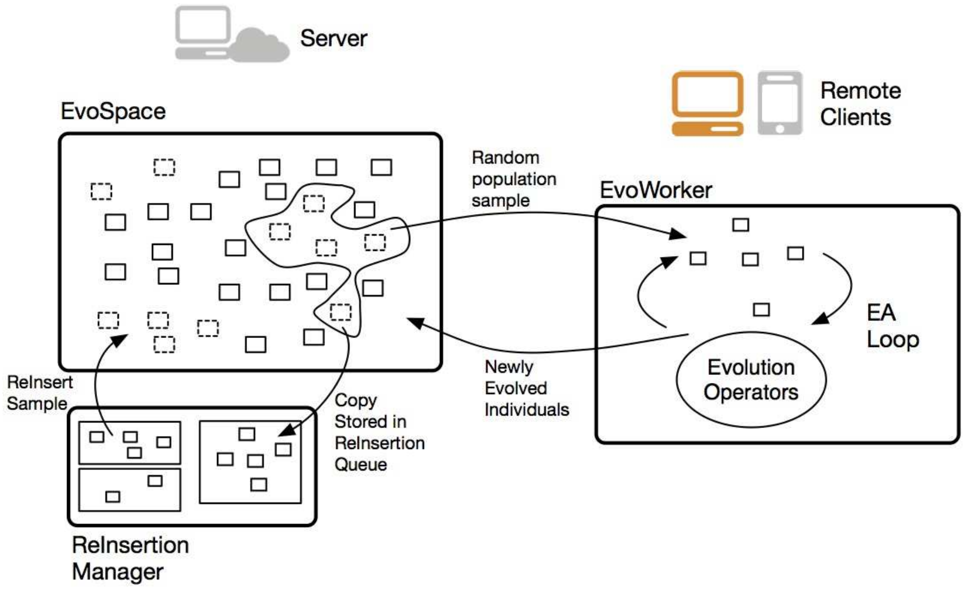

Figure 1: Main components and dataflow within EvoSpace.

the population size is below a certain threshold. Figure 1 illustrates the main components of EvoSpace.

3.1 Implementation

Populations of individuals are stored in-memory, using Redis, a key-value database, which was chosen over a relational database system, or other non-SQL alternatives, because it provides a hash based implementation of sets and queues which are natural data structures for the EvoSpace model. The logic of EvoSpace is implemented as a python module and exposed as a web service using Cherrypy http library. The EvoSpace modules are available with a Simplified BSD License from `anonymousURL`.

3.2 Evospace as a Heroku Application

Heroku (`http://heroku.com`) is a multi-language PaaS, supporting among others Ruby, Python and Java applications. The basic unit of composition on Heroku is a lightweight container running a single user-specified process. These containers, which they call *dynos*, can include web (only these can receive `http` requests) and worker processes (including systems used for database and queuing, for instance). These process types are the prototypes from which one or more dynos can be instantiated; if the number of requests to the server increases more instances can be assigned on-the-fly. In our case, our CherryPy web application server runs in one web process, when the number of workers was increased we added more dynos (instances) of the CherryPy process.

This model is very different from a VPS where users pay for the whole server; in a process based model, users pay only for the processes they need; being a *freemium* model means also that, if a minimum level of resources is not exceeded, it can be used for free.

Once deployed the web process can be scaled up by assigning more dynos; in our case and in the more demanding configurations of our experiments, the web process was scaled to 20 dynos. Instructions and code for deployment is available at `anonymousURL`

3.3 Evoworkers as PiCloud Jobs

PiCloud is a Platform as a Service (PaaS), with deep Python integration; using a library, Python functions are transparently uploaded to PiCLoud's servers as units of computational work they call *jobs*. Each job is added to a queue, and when there is a core available, the job is assigned to it. Both Heroku and PiCloud platforms are deployed on top of Amazon Web Services (AWS) infrastructure in the US-EAST Region. This ensures minimal latency and a high bandwidth communication between the services, and there is no charge for data transfer costs between both services. For the experiments type c1 and c2 Real Time workers where used. The code for the EvoWorkers implementation and experiment data is publicly available from a github repository `anonymousURL`.

4. EXPERIMENTAL WORK

4.1 Benchmark

The experiment reported here uses a multimodal problem generator. A P-Peaks generator has been chosen because the problem (and the computing resources needed for the search) can be appropriately scaled. Proposed by De Jong et al. in [**?**] a P-Peaks instance is created by generating a set of P random N-bit strings, which represent the location of the P peaks in the space. To evaluate an arbitrary bit string **x** first locate the nearest peak (in Hamming space). Then the fitness of the bit string is the number of bits the string has in common with that nearest peak, divided by N. The optimum fitness for an individual is 1. This particular problem generator is a generalization of the P-peak problems introduced in [**?**].

$$f_{P-PEAKS}(\mathbf{x}) = \frac{1}{N}\max_{i=1}^{P}\{N - hamming(\mathbf{x}, Peak_i)\} \quad (1)$$

A large number of peaks induce a time-consuming algorithm, since evaluating every string is computationally hard; this is convenient since to evaluate these type of distributed evolutionary algorithms fitness computation has to be significant with respect to network latency (otherwise, it would always be faster to have a single-processor version). However according to Kennedy and Spears [?] the length of the string being optimized has a greater effect in determining how easy or hard is the problem. In their experiments an instance having P = 200 peaks and N = 100 bits per string is considered to produce a considerably difficult problem.

4.2 Experimental Set-up

As EvoSpace is only the population store, EvoWorkers must implement the genetic operators. The genetic algorithm executed by EvoWorkers has been implemented using a modified DEAP (Distributed Evolutionary Algorithms in Python) framework [?] GA. Is important to note that only the basic non-distributed GA library was used. Three methods were added to the local algorithm: `getSample()` and `putBack()`; and another for the initialization of the population. The implementation of the local GA simply uses DEAPÂt's methods; for instance to generate the initial population, a local `initialize()` is called and the population sent to EvoSpace.

The selection of parameters was based on those used in [?]: a tournament size of 4 individuals, a crossover rate of 0.85 and a population of 512 individuals. In [?] a mutation rate equal to the reciprocal of the chromosome length; is recommended, as DEAP uses two parameters they were defined as follows, mutation probability of 0.5 and an independent flip probability of 0.02. For EvoWorkers the parameters were 128 worker generations for each sample, and a sample size of 16. To simulate unreliable workers each worker was assigned a return sample probability. In the experiments the lower probability was a 30% chance of an EvoWorker returning a sample or an EvoWorker failing 70% of the time; other return sample probabilities where 50%, 70% and 90%. Experiments where carried out for 4, 8 and 16 EvoWorkers. In a pool based asynchronous GAs there is usually no need to wait for a workerÂt's job to start a new generation. Although supported by EvoSpace time outs were not chosen as triggers to feed the population with new individuals, the population size was used instead. We believe the population size is a better threshold as it is more critical to the GA performance. In a previous work we found that when the population remaining in the pool was near starvation, the time of completion was increased. For these experiments, the insertion of individuals was triggered when less than 128 individuals remain in the population; the number of individuals feed to the population was 128, or 8 samples when the re-insertion algorithm was used. A summary of the setup is presented in table 1.

4.3 Results

In this section, results from the experiments are discussed. Figure 2 shows the time required to solution when using four EvoWorkers. For a population of 512 individuals and a sample size of 16, there is no difference in the time required to solution for percentages of 50% and above. Both re-insertion algorithms had comparable times. For 30 percent, both approaches had slight increase in time. For 8 workers (see Figure 3) there was marginal decrease in overall time; and

Table 1: GA and EvoWorker parameters for experiments.

GA Parameters	
Tournament size	4
Crossover rate	0.85
Population Size	512
Mutation probability	0.5
Independent bit flip probability	0.02
EvoWorker Parameters	
Sample Size	16
Generations	128
Other Parameters	
PiCloud Worker Type	Realtime
Number of Workers	4,8,16
Return Sample Probability	30%,50%,90%
Number of Executions	30

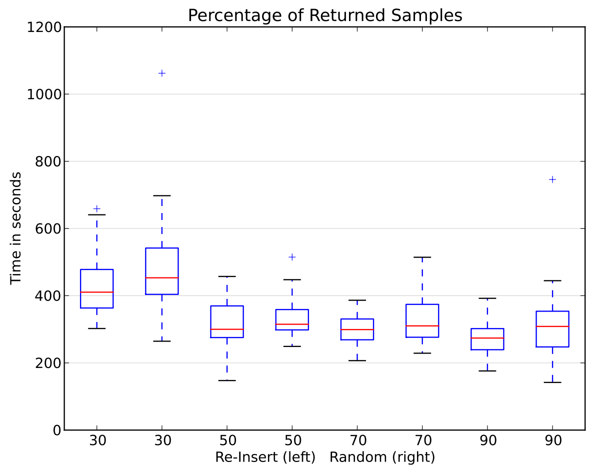

Figure 2: Time required to solution, 4 Workers

results where similar to those found in the experiments with 4 workers. Figure 4 shows results for 16 workers, when there was only a 30% chance of returning a sample the rate of re-insertions was high, approximately once every 35 samples. In this case, the insertion of random individuals resulted in a higher time to solution. For these experiments when there is not a high rate of re-insertion, both alternatives have similar results, but the re-insertion algorithm is better for situations when there are many starvation conditions. It appears that the insertion of random individuals is not detrimental when there are other evolved individuals in the pool. But when the remaining pool almost consists of random individuals, samples pulled by EvoWorkers need to start the search all over again. If not many samples are then returned to the pool, the work needed to reach an optimum is increased. Figure 5 also shows the number of evaluations needed to reach an optimum again for 16 workers. Figures 6 and 7 show the time required to solution for 30 and 90 percent of returned samples. For 90% both algorithms had similar speedups when

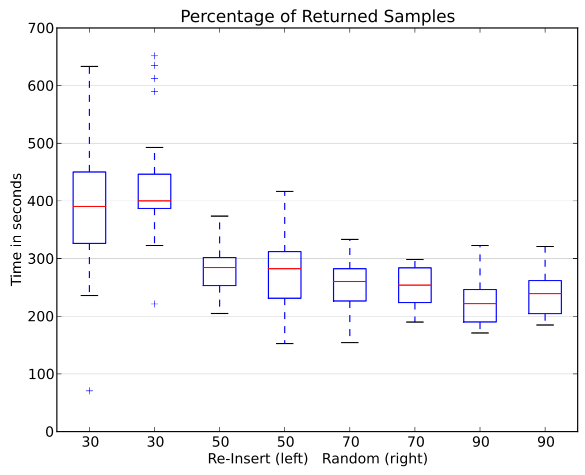

Figure 3: Time required to solution, 8 Workers

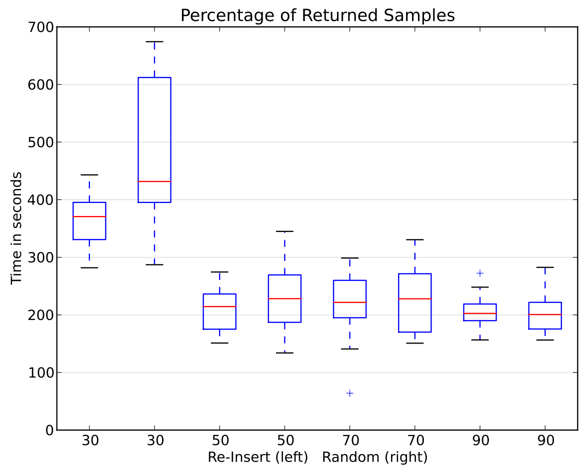

Figure 4: Time required to solution, 16 Workers

incrementing the number of workers. The marginal speedup obtained for these experiments is related to the population size, but this parameter was not changed. For 30% there was practically no speedup at all. The re-insert algorithm although not significant, had consistent decrease in time.

Fitness by time was measured as the average from each consecutive sample pulled by each worker. For each sample the average fitness was measured at the start and at the end of the local evolution. Also the minimum and maximum fitness values at start and finish was recorded. Final fitness values are shown in double width lines in figures. Readings used for these figures include all samples, including those that where not returned. Figure 8 shows fitness by time for the random insertion algorithm, as expected initial fitness drops at certain points, when random insertion occurs. Average final fitness is affected by random insertions.Figure 9 shows results for the re-insertion algorithm, with more characteristic curves for this type of algorithms.

5. CONCLUSIONS AND FURTHER WORK

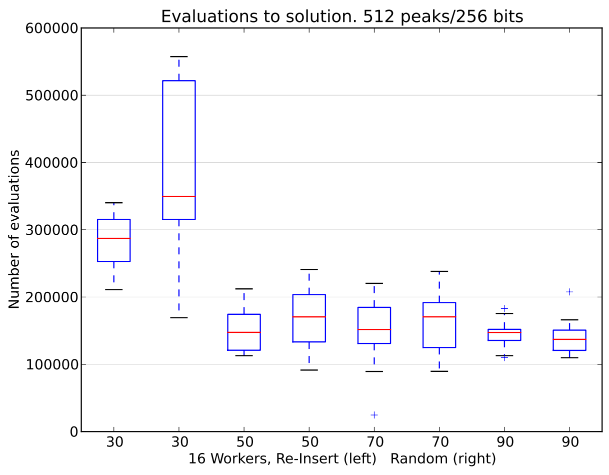

Figure 5: Number of evaluations needed to solution, 16 Workers

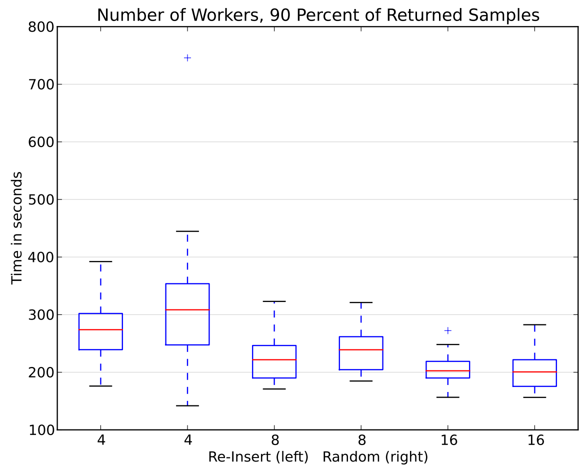

Figure 6: Time required to solution, 90% of returned samples

The re-insertion algorithm proposed for EvoSpace is a viable alternative to deal with a starving population in pool based GAs, and unreliable EvoWorkers. Using a benchmark problem from the P-Peaks problem generator, the approach was compared against the option of inserting randomly generated individuals. The same parameters and computing resources were used when testing both algorithms with better times reported when using the proposed technique. For experiments where the population size is enough for the number of workers with their sample sizes, plus a buffer to account for loss samples, then both algorithms could be used. However for cases when the number of EvoWorkers is unknown an hybrid approach could be used, insertion of random individuals to gradually increase the population size, and a re-insertion queue to handle lost samples. Further work could be focused on a hybrid approach, and also using heterogeneous computing resources.

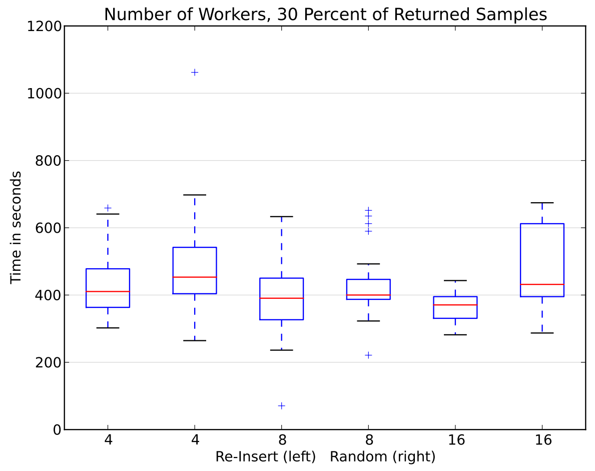

Figure 7: Time required to solution, 30% of returned samples

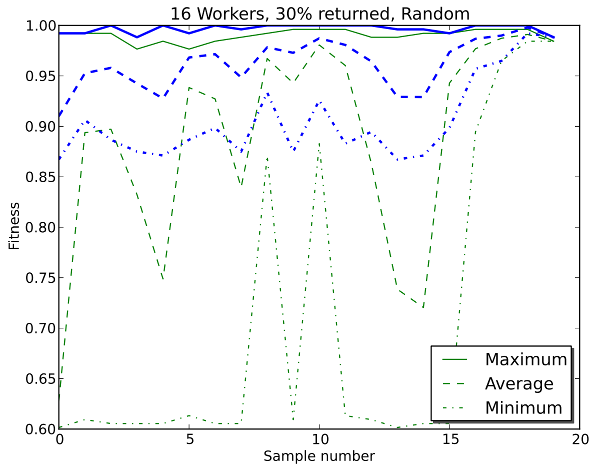

Figure 8: Fitness by Sample number, 30% of returned samples, 16 Workers, Random Algorithm

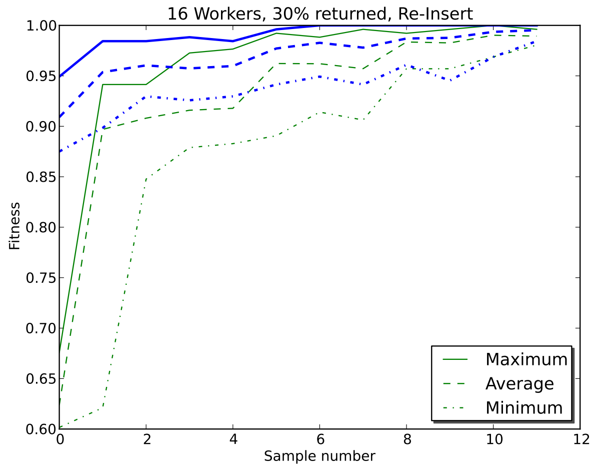

Figure 9: Fitness by Sample number, 30% of returned samples, 16 Workers, Re-Insert Algorithm

6. ACKNOWLEDGEMENTS

Hidden for double-blind review.